Enunciado Trabajo Práctico 2 - Listas

Delfina Martín, Federico Severino Guimpel Junio 2022

1. Contexto

El Censo Nacional de Población, Hogares y Viviendas es el recuento de todas las personas, todos los hogares y todas las viviendas que se encuentran en el territorio nacional en un momento determinado y se realiza aproximadamente cada 10 años. El objetivo es saber cuántas personas somos, cómo vivimos y cómo nos distribuimos. El Censo pregunta sobre sexo y edad, características del hogar y la vivienda, salud, educación, migraciones y ocupación, entre otros temas. Además, por primera vez en el censo 2022 se indagó por la identidad de género.

La información estadística que surge a partir del operativo sirve para diseñar políticas públicas esenciales en diversos ámbitos. También la utilizan las personas y empresas para investigar, planificar y llevar adelante proyectos basados en información oficial.

En este trabajo prctico se utilizarán los resultados del censo del 2010 debido a que los resultados del último censo no se encuentran todavía disponibles. A partir de la información provista se podrán aplicar transformaciones a los datos, filtrarlos, operarlos entre sí o alguna combinación de ellos.

2. Implementación de la estructura de datos

2.1. Entrada del programa

El programa leerá la información de un archivo cuyo nombre no deberá estar explícito en el código. La entrada formateada como una tabla luce así:

Provincia	Viviendas Particulares Habitadas	Viviendas Particulares Deshabitadas	Viviendas colectivas
Buenos Aires	4425193	952593	5750
Catamarca	89376	24258	385
Chaco	270133	42469	370
...	...	...	...

El programa deberá leer la entrada y guardar la información en una estructura de datos de tipo **Lista**. Podés elegir si hacerla simplemente enlazada, doblemente enlazada, circular simplemente enlazada o circular doblemente enlazada. Deberás explicar el por qué de la elección en el proyecto donde lo creas más conveniente.

> El archivo adjunto está en extensión csv. **csv**, *comma separated values* es un formato que se utiliza para representar grandes cantidades de datos. Como su nombre lo indica, cada valor dentro de una fila está separado por una coma. Tenelo en cuenta para el procesamiento de entrada de tu programa.

2.2. Datos de la lista

Teniendo en cuenta la definición de nodo vista en clase para listas simplemente enlazadas (o su alternativa doblemente enlazada)

```
typedef struct _SNodo {
  int dato;
  struct _SNodo *sig;
} SNodo;
```

1) modificá el/los campos de datos para que puedan contener la información del archivo de entrada.

2) reimplementá las funciones en la implementación de listas para que sean consistentes con los nuevos campos de datos. Recordá evitar que las funciones pierdan generalidad: la idea es que la implementación relativa a los datos de este trabajo práctico no 'ensucie' la implementación de listas. **Ayuda**: utilizá funciones como argumento que permitan modificar el/los campos de datos.

3. Funciones de alto orden

Las funciones de alto orden en matemática y ciencias de la computación son funciones que reciben funciones como argumentos y/o devuelven funciones como resultado. En este curso nos centraremos en las funciones (de alto orden) que reciben funciones como argumento. En particular, las que veremos permiten capturar patrones comunes sobre listas.

Es común que cuando usamos listas querramos:

- **transformar** los elementos de la lista. Es decir, modificar cada uno de los elementos aplicandole una función a cada uno de ellos.
- **filtrar** los elementos de una lista. Es decir, obtener una nueva lista posiblemente con menos elementos que la primera en donde algunos elementos se eliminaron debido a que no pasaron la función filtro.
- **operar** los elementos de una lista. Es decir, aplicar una función operador que permita calcular resultados a partir de los elementos de la lista.

o una combinación de ellos.

Tema 0

La función de alto orden **map** nos permite **transformar** cada elemento de una lista a partir de una función que recibe como parámetro.

```
*SNodo map_listas(*SNodo lista, Transformacion t);
```

(o su análogo para listas doblemente enlazada) donde Transformacion es

```
typedef *SNodo (*Transformacion) (*SNodo nodo);
```

3) Implementá la función map de forma que no se modifique la lista argumento.

4) Calculá la cantidad de viviendas particulares por provincia.

Hay veces que nos interesa aplicar una transformación a los elementos de una lista solamente si pasan un filtro.

```
*SNodo filter_map_listas(*SNodo lista, Transformacion t, Filtro f);
```

(o su análogo para listas doblemente enlazada) donde Filtro es

```
typedef int (*Filtro) (*SNodo nodo);
```

5) Implementá la función filter_map sin modificar la lista original.

6) ¿Cuáles son las ventajas y desventajas de reutilizar el apartado 3) para el apartado 5)?

7) Calculá la cantidad total de viviendas por provincia de aquellas que tengan más de 1000 viviendas colectivas.

8) Sabiendo que los datos están ordenados alfabeticamente por provincia, ¿cuántos nodos tiene que recorrer en el peor caso una función que busca la cantidad de viviendas totales de una provincia en particular? ¿Qué ventajas tiene poder asumir que la lista está ordenada?

Tema 1

La función de alto orden **filter** nos permite **filtrar** elementos de una lista según el resultado de aplicar un **predicado**. Se llama **predicado** a las funciones que retornan valores booleanos (en el caso de C enteros 0 o 1 de forma equivalente).

```
*SNodo filter_listas(*SNodo lista, Predicado p);
```

(o su análogo para listas doblemente enlazada) donde Predicado es

```
typedef int (*Predicado) (*SNodo nodo);
```

3) Implementá la función filter de forma que no se modifique la lista argumento.

4) Calculá el listado de provincias con más de 1000 viviendas deshabitadas.

Hay veces que nos interesa aplicar un filtro y luego operar los elementos que pasaron dicho filtro entre sí. En este trabajo nos vamos a restringir a operaciones que den resultados numéricos enteros.

```
*SNodo filter_fold_listas(*SNodo lista, Predicado p, Operador o);
```

(o su análogo para listas doblemente enlazada) donde Operador es

```
typedef int (*Operador) (*SNodo n1, *SNodo n2);
```

5) Implementá la función filter_fold sin modificar la lista original.

6) ¿Cuáles son las ventajas y desventajas de reutilizar el apartado 3) para el apartado 5)?

7) Calculá la cantidad total de viviendas de las provincias que tengan más de 1000 viviendas colectivas.

8) Sabiendo que los datos están ordenados alfabeticamente por provincia, ¿cuántos nodos tiene que recorrer en el peor caso una función que busca la cantidad de viviendas totales de una provincia en particular? ¿Qué ventajas tiene poder asumir que la lista está ordenada?

Tema 2

La función de alto orden **fold** nos permite **operar** elementos de una lista entre sí. En este trabajo nos restringiremos a operadores que den de resultado valores numéricos enteros.

```
int fold_listas(*SNodo lista, Operador o);
```

(o su análogo para listas doblemente enlazada) donde Operador es

```
typedef int (*Operador) (*SNodo n1, *SNodo n2);
```

3) Implementá la función fold de forma que no se modifique la lista argumento.

4) Calculá la cantidad de viviendas desocupadas en Argentina.

Hay veces que nos interesa transformar los elementos de una lista y luego operar estos nuevos datos entre sí.

```
int map_fold_listas(*SNodo lista, Transformacion t, Operador o);
```

(o su análogo para listas doblemente enlazada) donde Transformación es

```
typedef *SNodo (*Transformacion) (*SNodo nodo);
```

5) Implementá la función map_fold sin modificar la lista original.

6) ¿Cuáles son las ventajas y desventajas de reutilizar el apartado 3) para el apartado 5)?

7) Calculá la cantidad total de viviendas de todo tipo en Argentina.

8) Sabiendo que los datos están ordenados alfabeticamente por provincia, ¿cuántos nodos tiene que recorrer en el peor caso una función que busca la cantidad de viviendas totales de una provincia en particular? ¿Qué ventajas tiene poder asumir que la lista está ordenada?

4. Criterio de evaluación

Se evaluará si:

- el nombre del archivo de entrada no está incluído explícitamente en el código.
- se cargó y se leyó correctamente la entrada. En particular si se evitó releer el archivo varias veces.
- se pidió memoria dinámica. Se analizará en detalle si hay pérdidas de memoria y/o lecturas y escrituras erróneas.
- se eligió adecuadamente cómo representar los datos habitacionales.
- se implementaron correctamente las funciones de alto orden.
- se utilizaron correctamente las funciones de alto orden.
- la calidad del código es buena. En particular se tendrán en cuenta todas las buenas prácticas vistas en clase incluyendo abstracción, la documentación de las funciones (signatura y declaración de propósito), generalización, simplificación de proposiciones lógicas, inclusión de constantes, etc.
- la calidad del proyecto es buena. En particular la modularización, la separación del proyecto en archivos si corresponde, la inclusión de un README, la inclusión de un makefile, etc.

En general, entre más robusto sea el programa, más puntaje se tendrá en calidad de código siempre y cuando contemplar más casos no haga ilegible el programa.

Cada una de estas **categorías** tendrá una **ponderación diferente**, es decir, no todas pesarán lo mismo para calcular la nota final.

Importante No se aceptarán programas que en su compilación y/o ejecución indiquen que se está escribiendo en espacios no permitidos de la memoria (segmentation fault).